Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 3

****

1 Introduction

In this lab, we will continue practicing SQL queries related to aggregate functions and joins on a Sakila as well as University schema. The sakila schema is given below:

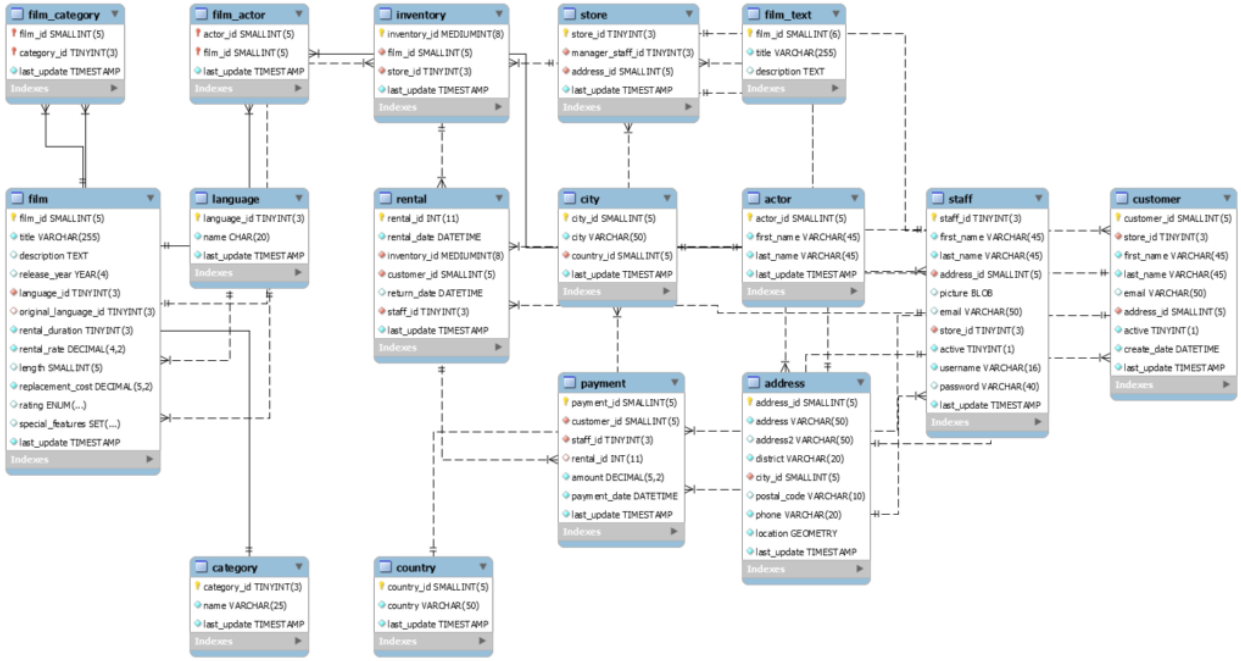


Figure:1 Sakila Schema

2 SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column. The aggregate function when used in select clause, computation applies to set of all rows selected. They are commonly used in combination with GROUP BY and HAVING clauses.

MySQL's aggregate function is used to perform calculations on multiple values and return the result in a single value like the average of all values, the sum of all values, and maximum & minimum value among certain groups of values. We mostly use the aggregate functions with SELECT statements in the data query languages.

The following are the syntax to use aggregate functions in MySQL:

```
function_name (DISTINCT | ALL expression)
```

In the above syntax, we had used the following parameters:

1. First, we need to specify the name of the aggregate function.
2. Second, we use the DISTINCT modifier when we want to calculate the result based on distinct values or ALL modifiers when we calculate all values, including duplicates. The default is ALL.
3. Third, we need to specify the expression that involves columns and arithmetic operators.

There are various aggregate functions available in MySQL. Some of the most used aggregate functions are summarized in the below table:

Aggregate Function	Descriptions
count()	It returns the number of rows, including rows with NULL values in a group.
sum()	It returns the total summed values (Non-NULL) in a set.
average()	It returns the average value of an expression.
min()	It returns the minimum (lowest) value in a set.
max()	It returns the maximum (highest) value in a set.
group_concat()	It returns a concatenated string.

Examples:

Find out the average of credits from course table in university database.

```
select avg(course.credits) as 'Avg Credit'
from university.course;
```

Find out the maximum and minimum value for credits from course table in university database.

```
select max(course.credits) as 'Max Credit',
min(course.credits) as 'Min Credits'
from university.course;
```

Find out the number of courses in the course table in university database.

```
select count(course.credits) as '# Course'
from university.course;
```

2.1 Removing Repeating Data with DISTINCT before Aggregation

How do aggregate functions handle repeated values? By default, an aggregate function includes all rows, even repeats, with the noted exceptions of NULL. We can add the DISTINCT qualifier to remove duplicates prior to computing the aggregate function.

Count number of different departments and number of total courses in the course table.

```
select count(course.title) as 'NoCourse', count(distinct
course.dept_name) as 'NoDept'
from university.course;
```

2.2 Group Aggregation Using GROUP BY

Aggregate functions return a single value, so we usually cannot mix attributes and aggregate functions in the attribute list of a SELECT statement.

Find out the average course credits and number of courses for each department in course table.

```
select course.dept_name, avg(course.credits) as 'Avg Credit'
from university.course;
```

The above query results in error. Because there are many departments and only one average course credit, SQL doesn't know how to pair them.

The following query will mix literals and aggregates.

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course group by course.dept_name;
```

2.3 Removing Rows before Grouping with WHERE

We may want to eliminate some rows from the table before we form groups. We can eliminate rows from groups using the WHERE clause.

Example:

Find number of courses and average credits for all course in Math department from course table.

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course
where course.dept_name = 'Math'
group by course.dept_name;
```

2.4 Sorting Groups with ORDER BY

We can order our groups using ORDER BY. It works the same as ORDER BY without grouping except that we can now also sort by group aggregates. The aggregate we use in our sort criteria need not be an aggregate from the SELECT list.

Example:

Find out the average course credits and number of courses for each department in course table and sort them by name of department

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course
group by course.dept_name
order by course.dept_name;
```

2.5 Removing Groups with HAVING

Use the HAVING clause to specify a condition for groups in the final result. This is different from WHERE, which removes rows before grouping. Groups for which the HAVING condition does not evaluate to true are eliminated. Because we're working with groups of rows, it makes sense to allow aggregate functions in a HAVING predicate.

HAVING: The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

A. If a GROUP BY clause is specified, the HAVING clause is applied to the groups created by the GROUP BY clause.

B. If a WHERE clause is specified and no GROUP BY clause is specified, the HAVING clause is applied to the output of the WHERE clause and that output is treated as one group.
C. If no WHERE clause and no GROUP BY clause are specified, the HAVING clause is applied to the output of the FROM clause and that output is treated as one group.

Find out the average course credits and number of courses for each department having average course credits more than or equal to 3.5 in course table and sort them by name of department.

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course
group by course.dept_name
having avg(course.credits) >=3.5
order by course.dept_name;
```

3 Join – Retrieve data from multiple Tables

3.1 SELF Join

The self-join is used to join a table to itself when using a join.

A self-join is useful for when you want to combine records in a table with other records in the same table that match a certain join condition.

Consider the following example:

To retrieve all customers whose last name matches the first name of another customer. We achieve this by assigning aliases to the customer table while performing an inner join on the two aliases. The aliases allow us to join the table to itself because they give the table two unique names, which means that we can query the table as though it was two different tables.

```
SELECT
  a.customer_id,
  a.first_name,
  a.last_name,
  b.customer_id,
  b.first_name,
  b.last_name
FROM customer a
INNER JOIN customer b
```

```
ON a.last_name = b.first_name;
```

Self joins aren't limited to the INNER JOIN. You can also use a LEFT JOIN to provide all records from the left "table" regardless of whether there's a match on the right one.

Try the following query:

```
SELECT
  a.customer_id,
  a.first_name,
  a.last_name,
  b.customer_id,
  b.first_name,
  b.last_name
FROM customer a
LEFT JOIN customer b
ON a.last_name = b.first_name
ORDER BY a.customer_id;
```

And of course, you can also use a RIGHT JOIN to provide all records from the right "table" regardless of whether there's a match on the left one.

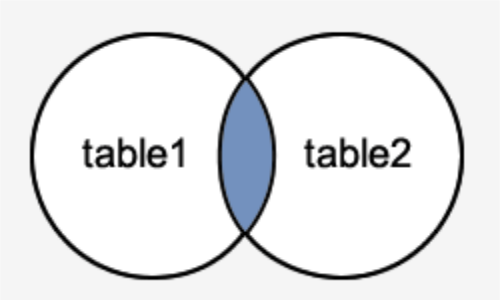
Try the following query:

```
SELECT
     a.customer_id,
     a.first_name,
     a.last_name,
     b.customer_id,
     b.first_name,
     b.last_name
FROM customer a
RIGHT JOIN customer b
ON a.last_name = b.first_name
ORDER BY b.first_name;
```

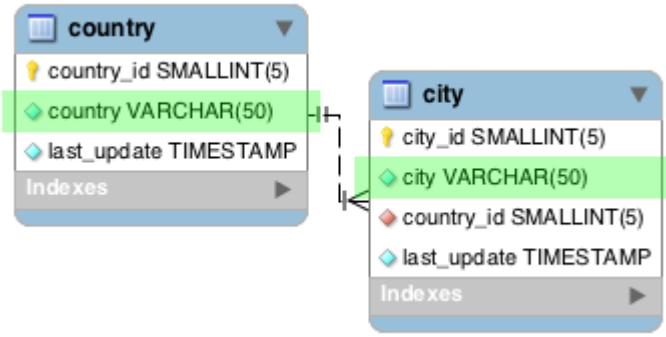
3.2 INNER Join

The INNER JOIN is used to return data from multiple tables. More specifically, the INNER JOIN is for when you're only interested in returning the records where there is at least one row in both tables that match the join condition.

We can understand it with the following visual representation where Inner Joins returns only the matching results from table1 and table2:



Consider the following tables in Sakila database:

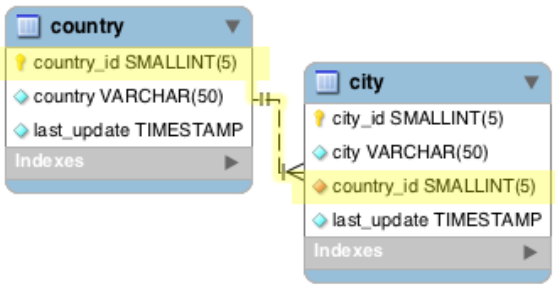


If we want to select data from the two highlighted fields (country and city), we could run the following query (which includes an inner join):

```
SELECT city, country
FROM city
INNER JOIN country ON
city.country_id = country.country_id;
```

In the above example, we use an inner join to display a list of cities alongside the country that it belongs to. The city info is in a different table to the country info. Therefore, we join the two tables using the country_id field — as that is a common field in both tables (it's a foreign key field).

Here's a diagram of those two tables (with the foreign key relationship highlighted).



Inner Joins with GROUP BY and Aggregate Functions:

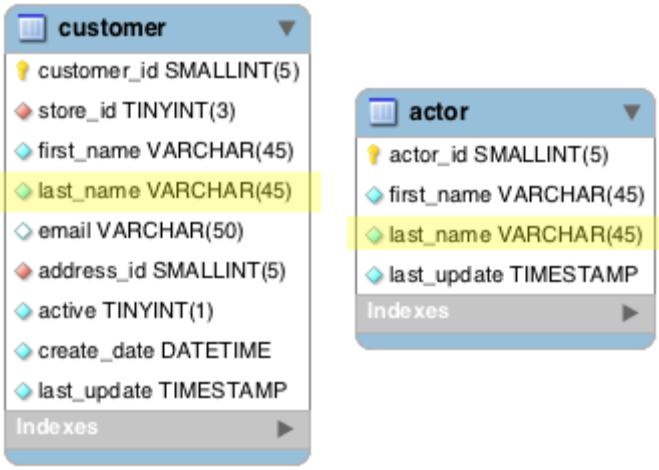
In the following example, we switch it around and provide a list of countries in one column, with the number of cities that each country contains in another column.

```
SELECT country, COUNT(city)
FROM country a
INNER JOIN city b
ON a.country_id = b.country_id
GROUP BY country;
```

3.3 Left Join

The LEFT JOIN is used to return data from multiple tables. In particular, the "LEFT" part means that all rows from the left table will be returned, even if there's no matching row in the right table. This could result in NULL values appearing in any columns returned from the right table.

Consider the following tables:



Let's return a list of all customers (from the customer table). And if the customer shares the same last name with an actor (from the actor table), let's display that actor's details too.

But the important thing is that we display all customers — regardless of whether they share their last name with an actor. Therefore, if a customer doesn't share the same last name as an actor, the customer is still listed.

We could achieve that with the following query:

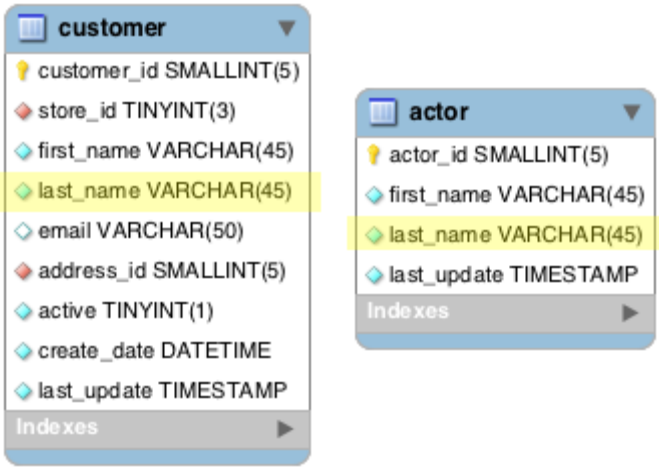
```
SELECT
     c.customer_id,
```

```
    c.first_name,
    c.last_name,
    a.actor_id,
    a.first_name,
    a.last_name
FROM customer c
LEFT JOIN actor a
ON c.last_name = a.last_name
ORDER BY c.last_name;
```

3.4 Right Join

The RIGHT JOIN is used to return data from multiple tables. In particular, the "RIGHT" part means that all rows from the right table will be returned, even if there's no matching row in the left table. This could result in NULL values appearing in any columns returned from the left table.

Consider the following tables:



Let's return a list of all actors (from the actor table). And if the actor shares the same last name with a customer (from the customer table), let's display that customer's details too.

But the important thing is that we display all actors — regardless of whether they share their last name with a customer. Therefore, if an actor doesn't share the same last name as a customer, the actor is still listed.

We could achieve that with the following query:

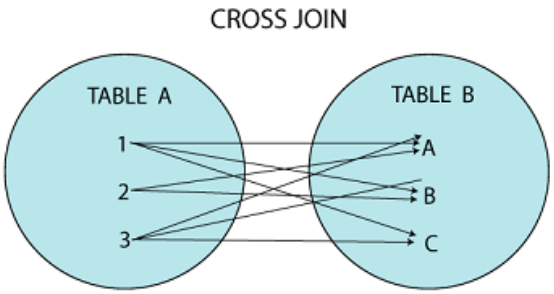
```
SELECT
    c.customer_id,
    c.first_name,
```

```
    c.last_name,
    a.actor_id,
    a.first_name,
    a.last_name
FROM customer c
RIGHT JOIN actor a
ON c.last_name = a.last_name
ORDER BY a.last_name;
```

3.5 CROSS Join

MySQL CROSS JOIN is used to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables. The Cartesian product can be explained as all rows present in the first table multiplied by all rows present in the second table. It is like the Inner Join, where the join condition is not available with this clause.

We can understand it with the following visual representation where CROSS JOIN returns all the records from table1 and table2, and each row is the combination of rows of both tables.



The CROSS JOIN keyword is always used with the SELECT statement and must be written after the FROM clause. The following syntax fetches all records from both joining tables:

```
SELECT column-lists
FROM table1
CROSS JOIN table2;
```

Example: Try the following query. It returns all rows from course table, repeating each row for number of rows in department table.

```
select * from university.course cross join
university.department;
```

3.6 NATURAL Join

When we combine rows of two or more tables based on a common column between them, this operation is called joining. A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type. It is like the INNER or LEFT JOIN, but we cannot use the ON or USING clause with natural join as we used in them.

Points to remember:

A. There is no need to specify the column names to join.
B. The resultant table always contains unique columns.
C. It is possible to perform a natural join on more than two tables.
D. Do not use the ON clause.

The following is a basic syntax to illustrate the natural join:

```
SELECT [column_names | *]
FROM table_name1
NATURAL JOIN table_name2;
```

Example:

Show the list of courses from table course along with department details from department table.

```
select * from university.course
natural join university.department;
```

4 UNION and UNION ALL

MySQL Union clause allows us to combine two or more relations using multiple SELECT queries into a single result set. By default, it has a feature to remove the duplicate rows from the result set.

Table 1	
Column 1	Column 2
A	1
A	2
A	3

∪

Table 2	
Column 1	Column 2
A	1
B	2
C	3

=

Table 1 Union Table 2	
Column 1	Column 2
A	1
A	2
A	3
B	2
C	3

Union clause in MySQL must follow the rules given below:

A. The order and number of the columns must be the same in all tables.
B. The data type must be compatible with the corresponding positions of each select query.
C. The column name in the SELECT queries should be in the same order.

Syntax:

```
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2;
```

Example:

```
select * from university.course
union
select * from university.department;
```

The above query gives an error because name and number of columns are different in both tables. The UNION clause combine data of two or tables of same type and remove duplicates.

```
select * from university.course
union
select * from university2.course;
```

The above query combine data of course table from two different university.

This operator returns all rows by combining two or more results from multiple SELECT queries into a single result set. It does not remove the duplicate rows from the result set.

We can understand it with the following pictorial representation:

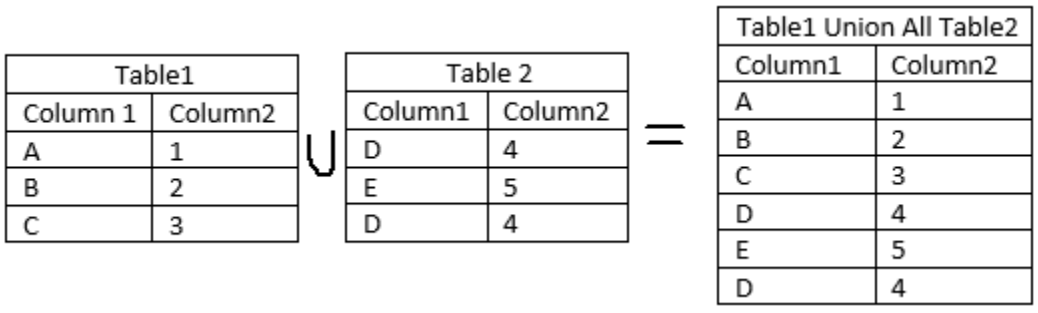


Table1	
Column 1	Column2
A	1
B	2
C	3

∪

Table 2	
Column1	Column2
D	4
E	5
D	4

=

Table1 Union All Table2	
Column1	Column2
A	1
B	2
C	3
D	4
E	5
D	4

5 Points to remember

- ✓ The difference between Union and Union All operators is that "Union" returns all distinct rows (eliminate duplicate rows) from two or more tables into a single output. In contrast, "Union All" returns all the rows, including duplicates rows.
- ✓ In MySQL, JOIN, CROSS JOIN, and INNER JOIN are syntactic equivalents (they can replace each other). In standard SQL, they are not equivalent. INNER JOIN is used with an ON clause, CROSS JOIN is used otherwise.
- ✓ A USING clause can be rewritten as an ON clause that compares corresponding columns. However, although USING and ON are similar, they are not quite the same. Consider the following two queries:

```
a LEFT JOIN b USING (c1, c2, c3)
```

```
a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3
```

With respect to determining which rows satisfy the join condition, both joins are semantically identical.

With respect to determining which columns to display for SELECT * expansion, the two joins are not semantically identical. The USING join selects the coalesced value of corresponding columns, whereas the ON join selects all columns from all tables. For the USING join, SELECT * selects these values:

```
COALESCE(a.c1, b.c1), COALESCE(a.c2, b.c2), COALESCE(a.c3, b.c3)
```

For the ON join, SELECT * selects these values:

```
a.c1, a.c2, a.c3, b.c1, b.c2, b.c3
```

With an inner join, COALESCE(a.c1, b.c1) is the same as either a.c1 or b.c1 because both columns have the same value. With an outer join (such as LEFT JOIN), one of the two columns can be NULL. That column is omitted from the result.

Exercise

1. Write the following queries in SQL, using the sakila schema.
    a. Retrieve a list of film titles along with the names of the actors in each film.
    b. Find the total amount paid by each customer.
    c. Identify films that were not rented out in January 2005.
    d. Calculate the total revenue of each store.
    e. List all customers who have rented a film in the last 30 days.
    (Hint: You can use 'INTERVAL x DAY' to specify an interval in MySQL.)

2. Fill-in-the-Blanks:
    a. The _____ statement is used to remove a table schema and its data from the database.
    b. In SQL, _____ is a constraint that ensures all values in a column are different.
    c. The _____ statement is used to add, delete, or modify columns in an existing table.
    d. To remove duplicates from the result set, you should use the _____ keyword with your SELECT statement.
    e. A _____ key is a field in one table that uniquely identifies a row of another table or the same table.

3. Answer the following Questions:
    a. Explain the difference between the `WHERE` and `HAVING` clauses in SQL.
    b. Describe the difference between `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL OUTER JOIN`.
    c. How do aggregate functions work in SQL when used with the `GROUP BY` clause?
    d. Can you use aggregate functions in the `SELECT` statement without `GROUP BY` clause? If so, what is the outcome?
    e. What are the performance implications of using `JOIN` operations in SQL, and how can you mitigate potential issues?

**********END**********